INDUSTRY STUDIES ASSOCIATION

INDUSTRY STUDIES ASSOCATION
WORKING PAPER SERIES

Is Open Source Software "Better" Than Closed Source Software? Using Bug-Fix Rates to Compare Software Quality

By

Jennifer Kuan
Software Industry Center
Carnegie Mellon University
Pittsburgh, PA 15213
and
Institute for Economic Policy Research
Stanford University
Stanford, CA 94305

2004
Industry Studies Association
Working Papers

WP-2004-11
http://isapapers.pitt.edu/

Is Open Source Software "Better" Than Closed Source Software? Using Bug-Fix Rates to Compare Software Quality

Jennifer Kuan
Stanford Institute for Economic Policy Research
Stanford University, 94305
650-724-4371
jwkuan@stanford.edu

October 2, 2003

Abstract:

If free software is also better software, as is claimed, we should all be downloading programs one day. While software quality itself is hard to define and measure, one can compare the *rate of change* of quality of open and closed source programs by comparing three pairs of comparable programs. Estimates of a hazard rate model find that the open source version of two programs are modified more quickly in response to bug reports. The two "faster response" open source programs are supported by communities of user-developers, whereas a third "slower response" program was written by programmers who were paid to develop the software for others. Thus open source software is found to undergo improvements more rapidly when it is produced and maintained by its users.

Acknowledgements: I would like to thank Tim Bresnahan, Paul David, Paul Gertler, Bronwyn Hall, Ben Hermalin, Aija Leiponen, David Mowery, and Rosemarie Ziedonis for their many helpful suggestions.

Key words: open source, software, modularity

I. Introduction

Open source software is free and accessible, easily downloaded from the Internet. Open source software is also flexible because the program is in its human readable form, or "source code". But is it also better than closed source software, as so many open source enthusiasts claim? For if this free and flexible software is also better than costly, inflexible closed source software, then significant changes are in store for software users and developers, great and small. Even now, IT (information technology) professionals must weigh the benefits of going with open source programs instead of proprietary source programs, computer science students ponder their future job market, and ordinary users wonder what will become of their favorite proprietary programs.

Likewise, software firms' reactions to upstart open source programs have ranged between extremes. Microsoft initially dismissed open source as irrelevant—a chaotic effort by amateurs; and the failure of open source Star Office to challenge Microsoft Office seems to bear this view out. But open source successes like Apache and Linux have gained ground against closed source incumbents Netscape and Solaris. Microsoft now warns of the dangers of open source software while Apple Computer uses an open source "kernel" in its once famously proprietary operating system.

This wide range of reactions to open source programs reflects the complexity and uncertainty regarding open source competitiveness. Is open source so much better than closed source that it is the future of software? Of course, software quality is difficult to

define let alone measure. Nevertheless, I consider three pairs of open and closed source programs and compare their *rates* of improvement; i.e., how quickly code gets changed. While rate of change is not itself a measure of absolute quality levels, it is an important aspect of quality even where static levels are easily defined. And by measuring rates of change, I avoid having to define and weight quality dimensions (is speed more important than flexibility; should portability be weighted more than ease-of-use?).

To measure the rate of change, I look at software bugs. Software bugs are commonly thought of as errors, but in fact comprise requests for information and new features as well. Thus "fixing bugs" is the mechanism by which software improves. I use a hazard rate model (the same sort of model used to calculate life expectancies) to compare the rates at which open source and closed source projects resolve service requests; i.e., how quickly they fix bugs.

I find that in two of three cases, open source communities address service requests more quickly than comparable closed source programs. What explains this? The popular debate and scholarly research offer a couple of hypotheses. One possibility is that open source code is designed to be more modular, which requires less coordination among programmers and which therefore allows broader participation among independent programmers. If this explained open source superiority, however, then all three open source programs should have been "better" than their closed source counterparts.

Instead, a second explanation fits the data better: that open source programs are better than closed source programs because programmers write the software for their own use. This hypothesis follows from a more general, theoretical model of user-production, but also has the simple intuitive interpretation that if a programmer is writing something for himself, he'll naturally work harder and do a better job. Thus, we observe that the two "faster response" open source programs are organized by users writing code for their own use, as is typical for open source projects, while the third, "slower response", program is written for others to use, a model more common to closed source projects.

II. What Is Open Source Software?

Software developers write human-readable source code and then compile that code into machine-readable object code. When we "buy" a typical program from a software company, we actually get the object code on a CD and the right to execute that object code. Since the object code is a set of ones and zeroes to be read by a computer processor, a user cannot possibly modify the program.

Open source software is not only downloadable for free on the Internet, it is also available in its human-readable form, source code. Thus open source software has also been called "freeware" because it can be freely downloaded, and because the users who download the software are "free" to modify and use the software however they like. The term "open source" therefore refers to the source code and distinguishes the open source

mode of software development from closed source development, where object code is kept proprietary.[1]

The "free and open source" model has been strongly influenced by an early proponent of open source software, Richard Stallman who founded GNU in 1984. Stallman (1999) began his experiment in open source software while working at the MIT computer lab. He wrote a printer driver program because his printer did not work with his computer. He reasoned that if the printer's manufacturer had made the printer driver's source code available, he could have modified it rather than written a whole new program. Ultimately, he decided that copyrights[2] restrict users' freedom to use and modify software. In response, Stallman created the GNU General Public License (GPL), which defines

[1] Note that open source software is different from "share-ware," software whose *object* code is downloadable free of charge. In some cases, individuals write programs and then post them on the Internet for others to use. Along with the program is often a request for $5 or an amount equal to the consumer's willingness to pay (e.g., McAfee Associates, see Shapiro and Varian, 1999, p. 90). S-ware is also a common marketing tactic. Software producers often put demonstration versions of their software on the Internet for free download. These "demo" versions are limited in functionality and allow prospective customers to evaluate the software before paying for it (Shapiro and Varian, 1999). See Takeyama (1994) for a discussion of the share-ware business model.

[2] A copyright is not infringed if a program accomplishes a task, using original, independently created source code. From the Gates Rubber Co. v. Bando Chemical Industries, Ltd. case (1993), "The main purpose or function of a program will always be an unprotectible idea," (Goldstein, 1997). Thus, a programmer can legally "copy" a copyrighted piece of software by independently recreating its functionality.

copylefted software as not restricting users' ability to use, distribute, or modify software and which requires modifications to be made publicly available.[3]

A stylized version of the modern practice of open source software has programs available for download on the Internet. Users download, use, and modify programs if they wish, and report bugs, questions, and requests for new features to a electronic bulletin board, also on the Internet. Modified software gets re-posted to the Internet for others to download and improve further. Improvements and innovations thus accumulate in the latest version of the software for everyone to use.

While this stylized version still holds for many users, open source software has, in some respects, "gone commercial." Entrepreneurs, such as Red Hat Software sell bundles of different stable open source programs called "distributions". These distributions get updated on a regular basis and are offered to users along with service contracts. And customers of open source software are now as likely to be corporate IT departments as individual users. In any case, programs like Linux, an operating system, and Apache, a web server program, have developed an enormous, world wide following with hundreds of contributors and hundreds of thousands of users. See the Netcraft survey for the latest figures on Apache users and Lee and Cole (2000) on Linux contributors.

---

[3] Heffan (1997) discusses the enforceability of this contract. Moglen (2003) argues that the GPL is easily enforced as a license to distribute software.

III. Debating the Merits of Open Source Software

By making good, flexible, free software available to all, open source software has thrilled users and alarmed software companies. An active public debate about the merits of open and closed source software and production has ensued. Theory informs this public debate and helps to form testable hypotheses.

A. Popular Claims

The conflicting claims about why open source software is good or bad involve several levels of analysis, though ultimately, the focus will be on the software product itself. On the social level, open source advocates argue that all software should be free. In Stallman's (1999) words, "The idea that the proprietary software social system—the system that says you are not allowed to share or change software—is antisocial, that it is unethical, that is simply wrong, may come a s a surprise to some readers. But what else could we say about a system based on dividing the public and keeping users helpless?"

Likewise, closed source proponents argue that at least some proprietary software is good for society. In Bill Gates' (Lettice, 2001) view, "I certainly don't agree with the…view that there should be no jobs in this area…The kind of commercial advances and risk taking that we've been able to do…you can't get things like speech recognition on a tablet computer coming out of that paradigm."

At the industry level, Microsoft also makes the more specific argument that the copyleft prevents closed source firms from using open source code because the it forces firms to disclose their own proprietary code—and this harms the entire software industry (Gomes, 2001).

Finally, at the product level, closed source and open source advocates have both taken up the claim that theirs is the better product. Usually, open source proponents focus on specific quality dimensions, such as robustness, portability and speed. These are dimensions in which closed source software is often particularly weak. For instance, Microsoft Windows is prone to frequent crashes; Windows users therefore find robustness (infrequent crashing) especially appealing. But closed source firms make the rather convincing case that open source projects are amateurish and unable to meet their customers' needs. The experience of open source Star Office seems to bear this out. Though marketed now by Sun Microsystems, Star Office has failed to challenge Microsoft Office, Microsoft's office productivity software (Mossberg, 2002).

B. Theories

Theories for *why* closed source software should be better than open source software are not generally made explicit because the closed source model is the incumbent, profit-maximizing model that, in theory, serves consumers—and economic efficiency—so well in general. That is, closed source firms pay professional software developers to serve customers, just as any other type of firm services its customers.

Given this, it is counterintuitive indeed that open source should be better than closed source; hence a surfeit of theories. An influential early theory is suggested by a practitioner, Raymond (1999), who argues that the structure of open source organization generates a better end-product. Closed source projects are organized centrally (like cathedrals) while open source projects are distributed (like bazaars). This decentralized, or modular, structure makes it possible for many autonomous individuals to get involved, so more bugs will be found and fixed, and more useful features added, than with closed source programs. As Raymond puts it, "Given enough eyeballs, all bugs are shallow."

This cathedral-bazaar notion has its antecedents in the scholarly literature on modularity, where the benefits of modularity and the organizational or industrial structure are articulated. For example, Langlois and Robertson (1995) argue that modularity is good for innovation because it allows many different approaches to be tried out simultaneously by autonomous, individual innovators; and Baldwin and Clark (2000) observe that modularity in the computing industry has allowed for increasingly complex products. Both of these arguments align with the popular notion of open source as many simultaneous actors working to create a high quality, complex piece of software.

Similarly, the literature also explores how modularity relates to organization. Baldwin and Clark (2000) describe how "visible design rules" serve as an interface among firms in a modularized industry and allow independent, autonomous firms to work together. Others describe how modularized products can and should be designed within a firm. For

instance, Sanchez and Mahoney (1996) argue that modularizing product design within the firm makes the firm more flexible and affects the way a firm learns. Sigglelkow and Levinthal (2003) consider the degree and timing of modularization and their implications for centralizing or decentralizing within the firm. And Garcia and Steinmueller (2003) introduce the idea whether modules relate horizontally or vertically affects how centralized or decentralized authority should be. Finally, whether modules should be designed by one firm or many firms is tackled by Langlois' (2002) exploration of firm boundaries.

Thus the focus on modularity in the popular debate is amply supported by the research literature, where the whys and the how-tos are discussed. There is, however, another reason why open source programs should be better than closed source programs: Kuan (2001) argues that consumers who organize themselves to produce a good are more efficient than profit-maximizing producers because they have private information about their own demand. Applied to open source software, this means that programmers who write open source code for their own use will just plain work harder and thus make it better than if they were writing code to sell to others.

In the end, the social question of whether open or closed source is better for society cannot be evaluated empirically. But perhaps other questions, of product quality and organizational structure, can be. I look next at these latter questions.

IV. Empirical test: focus on bugs

Ideally, we would like to compare the "quality" of an open source program with that of a similar closed source program, in a sort of head to head comparison. But quality can be difficult to define and measure. This is not to say that it cannot be done; Wheeler (2003) surveys empirical studies attempting to compare various dimensions of software quality including market share, reliability (uptime), performance (speed), scalability, security, and total cost of ownership. Often, these studies offer a variety of convincing measures—which is precisely the point! In addition to the many measures of quality within each dimension of quality, there are many dimensions of quality. How do we choose which dimension of quality to measure, or how to weight different dimensions? Some users want easy to use software, while others care more about speed.

Instead, we can avoid having to define quality levels by comparing the speed at which remediation of defects are made. This rate of improvement comparison would be interesting even if static quality could be defined and measured, and has the added advantage of being easy to measure using service request, or "bug", data.

Figure 1 illustrates the software improvement process, beginning with bug discovery, continuing with bug resolution, and finishing with the integration of bug fixes into the released version of software. The end-result of this process is the object of interest, "quality". Thus bugs are central to the mechanism by which software improves.

(Figure 1 about here)

Data from three pairs of programs are assembled: (1) an open source web server and a closed source web server, (2) an open source operating system and a closed source operating system, and (3) an open source user-interface and a closed source user-interface.[4] The question: How quickly do bugs get fixed? The data include when the bug was discovered, when it was resolved, a priority rating (how important it is that a bug get fixed), and a severity rating (how bad the problem is). To address the question statistically, a hazard rate model is used (the same sort of model as is used for life expectancy) because the question, "How quickly do bugs get fixed?" is essentially the same as, "How long does a bug live?"

In addition to the basic question of whether open source bugs get fixed more quickly than closed source bugs, the data allow us to examine hypotheses suggested by the popular debate, above. First, we can count the bugs and determine whether the many open source "eyes" find and fix more bugs, as Raymond predicts. Bugs must be registered in order for fixes to be integrated into new versions of the software. Thus any bugs that get fixed must first have been reported and recorded in the data. Second, the variety of programs in the sample allows us to assess whether certain types of programs are "better" as closed source programs than others, as closed source proponents suggest.

[4] The data from the proprietary source programs were provided by closed source software firms under the condition their identity not be disclosed.

Before getting into the data and analysis, I remark briefly on the comparability of open and closed source bug data. We, of course, hypothesize that open and closed source development are different. However, there are enough similarities between open and closed source processes that it makes sense to compare bug life expectancies. First, the types of software being produced are comparable. Linux and FreeBSD are open source Unix operating systems, Solaris and HPUX are closed source Unix operating systems. Both open and closed source Unix operating systems do largely the same thing. Hence the comparison between functionally similar programs: Unix operating system with Unix operating system, web server with web server, etc.

Second, the steps of the software improvement process are performed whether open source or closed source. This is perhaps not surprising since open source contributors are often professional software or hardware developers who carry over large-project practices from their closed source experience. Thus the same sorts of problems get identified as bugs for open and closed source, these problems get documented in similar fashion, and the fixes for bugs are similar for open and closed source programs. Garzarelli (2002) discusses the phenomenon of common professional skills among open source participants, Mockus, et al (2000) document the Apache process, and Lee and Cole (2000) discuss the Linux process.

Still, there are differences both in bug discovery and installation, the two steps on either side of bug resolution. Users find bugs in open source programs, while the closed source firm's employees look for bugs in closed source programs. Also, in open source

programs, bug fixes are released to users quickly (perhaps too quickly for some users (Cass, 2003)) while in closed source programs, fixes tend to be released more slowly in larger batches. These differences may affect how we interpret the results of our comparison between open and closed source bug fixing, but they do not invalidate such a comparison.

V. The Data

Bug databases for all three open source programs are available on the Internet. The data from Apache and FreeBSD are a census of bugs found over the life of the program, while the data from Gnome include only the bugs that were fixed over a rolling 12-month period. The closed source bug data come from proprietary sources. Table 1 lists the available data.

(Table 1 about here)

The data for web servers cover the life of open and closed source programs, both of which are of similar vintage: work on Apache started in early 1995 and on the closed source program six months later. Similarly, both open and closed source operating systems were spun off from the original Berkeley Unix, though at different times. All data for FreeBSD going back to 1994 were available, but for the closed source program, only data from the latest major revision were available. The data from the user-interface programs is less complete. Only data on fixed bugs are available from the open source

program, and little historical information about the closed source program is available, though it was most likely founded much earlier than the open source program.

In all six databases, each bug is assigned a priority rating and a severity rating by the bug's finder. Ratings are on a scale of one to three. Programs give guidance on how to assign ratings. For example, priority ratings include "high", "medium", and "low"; severity ratings are described as "critical", "serious", "non-critical", and "wish-list". Often, definitions further clarify ratings; for example, "critical" might be further defined as "important and no work-around". Consistency across projects is further aided by the fact that open source contributors are familiar with standard closed source industry practices for assigning ratings. Since high priority bugs might get fixed more quickly than low priority bugs, for example, I control for priority and severity when comparing bug fix rates.

Figures 2 and 3 summarize bug discovery for open and closed source web servers and operating systems over time.

(Figures 2 and 3 about here)

VI. Results

Notice that far from supporting Raymond's predictions that open source users' many eyes find more bugs, closed source bugs far outnumber open source bugs. Another interesting

observation is that closed source bug discovery appears more volatile than open source bug discovery. One possible reason is organizational: new, improved versions of software get released to the market according to a marketing schedule. Spikes in bug discovery might reflect this internal process.

A. Statistical Model

To compare bug resolution rates, I compare baseline hazards for the three pairs of programs. The hazard is assumed to be

$$h(t) = h_o(t) \exp (\beta_1 x_1 + \beta_2 x_2)$$

where h_o is the baseline hazard, x_1 and x_2 are dummies for priority and severity, and β_1 and β_1 are coefficients of x_1 and x_2. A high hazard rate indicates that bugs get fixed quickly, while a low hazard rate indicates that bugs get fixed slowly.[5] In the graphs that follow, a point on the graph is interpreted as the likelihood of being fixed (y-axis) conditional on having survived up to that point in time (x-axis).

[5] Note that the hazard rate, $\lambda(t)$, is related to the survivor function, S(t), and that either could be used in this analysis. In particular, the probability of failing at time $t = f(t) = \lambda(t) \bullet S(t)$.

B. Open Source and Closed Source Web Servers

Figure 4 shows the baseline hazard rates of open and closed source web server programs. The baseline hazard of the open source web server (darker line) is higher than the baseline hazard of the closed source program, suggesting that open source bugs are resolved more quickly than closed source bugs—after controlling for priority and severity.

(Figure 4 about here)

C. Open Source and Closed Source Operating Systems

Figure 5 graphs baseline hazards for open and closed source operating system, controlling for priority and severity. The baseline hazard curves for operating systems cross, with the open source curve (darker line) starting out above the closed source curve. While this indicates that the long-lived open source bugs are more stubborn than their closed source counterparts, for the most part, outside the tail, open source bugs get fixed more quickly. Note also that data from the closed source program do not date as far back as the open source program, so the open source data have a longer tail.

(Figure 5 about here)

D. Open Source and Closed Source Graphical User Interface Programs

Figure 6 shows the baseline hazard rates for open (darker line) and closed source user-interface programs, controlling for priority and severity. Here the closed source program has a higher baseline hazard rate than Gnome, suggesting that closed source bugs get fixed more quickly than open source bugs.

(Figure 6 about here)

E. Covariates: Priority and Severity

Bugs are assigned priority ratings and severity ratings at the time the bug is entered into the database by the person who discovers the bug. This early, one-time-only assignment of ratings is a potential source of error because what might seem at first to be an unimportant bug might later be deemed an important bug, for example. However, this error would affect all bugs, open and closed source.

Ratings are remarkably consistent across programs. A top priority bug gets a rating of 1; a low priority bug is rated 3. A bug that presents huge problems to the user is rated very severe, or 1 on the severity scale, while a bug that is merely annoying rates a 3 on the severity scale.

Table 2 summarizes the coefficients for priority and severity. Note that the open source web server and open source user interface databases did not make use of the Priority rating, i.e. all of the priority ratings were set at the default level. A coefficient of less than one means that the higher the priority, or the greater the severity, the more quickly the bug was resolved.

(Table 2 about here)

VII. Discussion and Conclusion

Is open source better than closed source? In two of three cases, the answer seems to be yes, at least by our narrowly defined measure of bug lifetime. Table 3 summarizes the graphs above.

(Table 3 about here)

What explains these results? Two possibilities are discussed above: (1) Open source modularity means more bugs are found and fixed, and (2) Programmers work harder when they're writing software for their own use.

In the case of modularity, it would be nice to show that the open source programs are more modular than closed source programs and that this increased modularity accounts for performance differences. Unfortunately, direct evidence that open source code is

more modular than closed source code is difficult to obtain. While the modularity of open source code could, in principle, be measured, closed source code is usually unavailable for measurement, so the source codes themselves cannot be compared. Certainly in the case of the three closed source programs in my sample, access to the source code is impossible.

However, indirect evidence of relative open source modularity is suggested by the case of Netscape *Communicator*. The pioneering Internet browser, *Communicator*. started out as a closed source program. But as Microsoft's *Internet Explorer* gained ground against *Communicator*, Netscape decided to compete by making *Communicator* open source, in a project named "Mozilla". The closed source code turned out to be "the worst kind of spaghetti code…As a result, the Mozilla project had to do a lot of undoing and repairing before it could advance" (Moody, 2002). Programmer anecdotes suggest that the formerly closed source code was difficult to modify, requiring lots of time to figure out how to make small changes. Rewriting of this code, the "undoing and repairing" included making the program more modular, and thus easier for newcomers to make changes and improvements.

Communicator-Mozilla offers a rare peek into the secretive world of closed source software and suggests that closed source programs are indeed structured differently than more modular open source programs. Certainly, this is true for the three programs in my sample. Apache, FreeBSD and Gnome are all open source programs that started out life as open source programs. Apache and FreeBSD enjoy a tremendous reputation among

open source programmers as being modular and easy to understand and modify. Similarly, Gnome was founded by an experienced open source contributor. Thus all three open source programs have impressive pedigrees as modular open source programs, and are likely to be more modular than their closed source, "spaghetti code" counterparts. So if modular code structure explains open source performance, then all three open source programs should outperform their closed source competition. Instead, only two of three programs did so.

Is open source then better because programmers are working for themselves, our second hypothesis? Evidence suggests that programmers are driven primarily by their own need for a program.

For example, Apache founder Behlendorf (1999, p. 158) recalls that "Apache started with a group of webmasters sharing patches to the NCSA web server, deciding that swapping patches like so many baseball cards was inefficient and error-prone…". These early user-developers were "webmasters", people charged with managing web servers, whose paid work required them to use Apache. Next, a survey of Linux developers (Hertel, et al, 2003) finds that the top reason for contributing is to "facilitate my daily work due to better software." Shah (2003) finds much the same in her analysis of open source community mailing lists and interviews: contributors enter the world of open source software because they need to use the software; later, they stay on because they enjoy the work.

Thus a “selfish” motivation seems to be quite important among open source contributors. In our sample, it certainly speaks to the experience of Apache (as suggested by testimonials like the one above) and FreeBSD (as suggested by interviews with founders and contributors). Both of these programs outperform their closed source competitors.

However, Gnome, our third and “slower” open source program, is different. Gnome founder, Miguel de Icaza, describes his interest in open source this way (Weber, 2000): “We are going to level the playing field. This is about helping the consumer. This is for everybody.” Notice the emphasis on the consumer and “everybody” rather than on his own use for the program. Rather than the typical open source “I do this for me,” Gnome is “I do this for you.” Ironically, this makes Gnome more like closed source than open source: Gnome programmers are paid by investors to create software for others to use. With open source software, programmers create software for themselves to use.

How does this organizational structure affect bug fixing speed? Theory suggests that users who organize and produce their own software can do better than closed source firms because they have private information about their own demand (Kuan, 2001). Only a for-profit firm with perfect information about users could get users to pay every cent they are willing to pay; most for-profit firms have far less information than that. But a user knows how much he is willing to pay—in terms of dollars or effort, and if he writes the software for himself, he puts in all of that. Described this way, it is not so surprising that a user would work a little harder to write a piece of code for himself than he would for someone else whose willingness to pay for the code is only guessed at.

VIII. Conclusion

Is open source better than closed source? I offer one way to answer that question empirically. While "better" is hard to define, especially for software, I suggest that whatever dimension of quality matters most to users, faster is better than slower. Thus the rate of change can be used to compare open and closed source programs. Using a hazard rate model on bug databases, I find that the answer is "sometimes".

The next question: Why is open source sometimes better than closed source and sometimes worse? First, the theory that many open source eyeballs will find and fix more bugs than closed source programmers is not supported by the evidence. At least by numbers alone, closed source programmers find and fix substantially more bugs than open source programmers. Second, the much-touted modularity of open source programs may facilitate participation among open source contributors, but does not alone explain open source superiority over closed source: while all three open source programs have impressive credentials as modular open source programs, only two of three outperform their closed source counterparts.

Instead, the software's intended user—which in this case translates to its organizational form—matters. That is, users writing software for their own use (typical of open source) perform better than programmers writing software for other people to use (typical of closed source). This much is predicted by the general theory that consumers who organize to produce a nonrival good for themselves use private information about their

own demand, information which no producer has access to. Thus what seemed to be a technical matter, of structuring code modularly to be worked on by autonomous, far-flung contributors, turns out to be an issue of organization and incentives.

This naturally leads to the next question: when do programmers write software for themselves and when do they not? Table 4 lists some of the most popular open source programs (as measured by rate of downloads)—noting that there are lots of open source programs that never attract any attention or contributions. The technical nature of these programs seems to supports the notion that programmers are "scratching their own itch", but it also suggests that open source might remain an option for programmers only, leaving Microsoft and other closed source firms to serve their mostly non-technical customers.

(Table 4 about here)

Finally, if open source is indeed a user-as-programmer phenomenon, then it has some surprising relatives, including enterprise software and research and development (R&D). In the under-studied enterprise software industry, firms sell business software to large companies. The software often runs on mainframe computers, and almost always requires much more customized programming. Thus, here, too, users must program (or hire programmers).

As for R&D, of which software development is a special case, other researchers have observed similarities between open source software and academic research (Tuomi, 2000; Dalle and David, 2003). Does the user-producer idea from open source apply to academic researchers, too? If so, the open and closed source relationship described in this paper may improve our understanding of the university- and industry-based R&D relationship that has interested us for so long.

References

Baldwin, Carliss Y. and Kim B. Clark, 1997, "Managing in an Age of Modularity," Harvard Business Review, September – October, 84-93.

Behlendorf, Brian, 1999, "Open Source as a Business Strategy," in Chris DiBona, Sam Ockman, and Mark Stone (Editors), Open Sources: Voices of the Open Source Revolution (O'Reilly and Associates, Sebastapol), 149-170.

Cass, Stephen, 2003, "Linux's Challenge to Unix," IEEE Spectrum, June, 15-16.

Dalle, Jean-Michel and Paul David, "The Allocation of Software Development Resources in 'Open Source' Production Mode," SIEPR Discussion Paper No. 02-27, Stanford.

Dalle, Jean-Michel and Nicolas Jullien, 2003, "'Libre' Software: Turning Fads into Institutions?" Research Policy 32:1, 1-180.

David, Paul, 2001, "From Keeping 'Nature's Secrets' to the Institutionalization of 'Open Science,'" mimeo.

Garcia, Juan Mateos and W. Edward Steinmueller, 2003, "Applying the Open Source Development Model to Knowledge Work," mimeo.

Garzarelli, Giampaolo, 2002, "Open Source Software and the Economics of Organization," mimeo.

Goldstein, Paul, 1997, Copyright, 2nd Edition. Little, Brown.

Gomes, Lee, 2001, "The Campaign Against Linux Is Uphill Battle for Microsoft," Wall Street Journal, June 14.

Heffan, Ira, 1997, "Copyleft: Licensing Collaborative Works in the Digital Age." Stanford Law Review 49.

Hertel, Guido, Sven Niedner and Stephanie Hermann, 2003, "Motivation of Software Developers in Open Source Projects: An Internet-based Survey of Contributors to the Linux Kernel," Research Policy 32(7): pp. 1159-1177.

http://apache.org/

http://freshmeat.net/

http://www.bugs.apache.org/index

http://www.bugs.gnome.org/

http://www.freebsd.org/support.html\#gnats

http://www.netcraft.com/survey/

Kiefer, Nicholas M., 1988, "Economic Duration Data and Hazard Functions." Journal of Economic Literature 26, 646-679.

Kuan, Jennifer, 2001, "The Phantom Profits of the Opera: Nonprofit Ownership in the Arts as a Make-Buy Decision," Journal of Law, Economics, and Organization, 17(2).

Lakhani, Karim and Eric von Hippel, 2000, "How Open Source software works: "Free" user-to-user assistance." MIT Sloan School of Management Working Paper #4117.

Langlois, Richard N., 2002, "Modularity in Technology and Organization," Journal of Economic Behavior and Organization 49, 19-37.

---------- and P. Robertson, 1995, "External Capabilities and Modular Systems," in Firms, Markets and Economic Change, Routledge, 68-105.

Lee, Gwendolyn K. and Robert E. Cole, 2000, "The Linux Kernel Development As A Model of Knowledge Creation." mimeo, University of California at Berkeley.

Mockus, Audris, Roy T. Fielding, and James Herbsleb, 2000, "A Case Study of Open Source Software Development: The Apache Server," Proceedings of the 22nd International Conference on Software Engineering, ACM, 263-272.

Moglen, Eben, 2003, "Freeing the Mind: Free Software and the Death of Proprietary Software," Fourth Annual Technology and Law Conference, University of Maine Law School, Portland, June 29.

Moody, Glyn, 2002, "Mozilla, Tortoise of Freedom," computer weekly, Reed Business Information UK, April 24.

Netcraft, http://www.netcraft.com/survey/

Raymond, Eric, 1999, "The Cathedral and the Bazaar."
http://www.tuxedo.org/~esr/writings/cathedral-bazaar/

---, 2000 "Homesteading the Noosphere." http://www.tuxedo.org/~esr/writings/cathedral-bazaar/

Sanchez, Ron and Joseph T. Mahoney, 1996, "Modularity, Flexibility, and Knowledge Management in Product and Organization Design," Strategic Management Journal 19, 64-76.

Shah, Sonali, 2003, “Understanding the Nature of Participation and Coordination in Open and Gated Source Software Development Communities,” unpublished dissertation, MIT.

Shapiro, Carl and Hal R. Varian, 1999, Information Rules. Harvard University Press.

Siggelkow, Nicolaj and Daniel A. Levinthal, 2003, “Temporarily Divde to Conquer: Centralized, Decentralized, and Reintegrated Organizational Approaches to Exploration and Adaptation,” forthcoming in Organization and Science.

Stallman, Richard, 1999, “The GNU Operating System and the Free Software Movement,” in Chris DiBona, Sam Ockman, and Mark Stone (Editors), Open Sources: Voices of the Open Source Revolution (O’Reilly and Associates, Sebastapol), 53-70.

Takeyama, Lisa, 1994, "Distributing Experience Goods by Giving Them Away: Shareware--Some Stylized Facts and Estimates of Revenue and Profitability," Economic Innovation and New Technology 3:2, 161-74.

Tuomi, Ilkka, 2000, “Learning from Linux: Internet, innovation and the new economy,” mimeo, University of California at Berkeley.

Weber, Thomas E. 2000, “Here’s a Plan to End Microsoft’s Dominance (No Lawyers Needed).” Wall Street Journal, May 15.

Wheeler, David, 2003, “Why Open Source Software/Free Software (OSS/FS)? Look at the Numbers!” http://www.dwheeler.com/oss_fs_why.html

Figure 1: Software Improvement Process

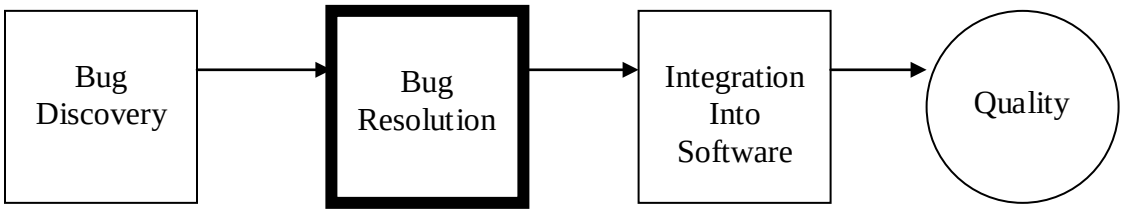

Table 1: Software Bug Data Availability

Program Type	Open Source Data	Closed Source Data
Web Servers	All bugs 3/96 – 12/99	All bugs 1/97 – 10/99
Operating Systems	All bugs 9/94 – 2/00	All bugs for a single version 1/97-6/99
User-Interfaces	All fixed bugs 2/99-1/00	All bugs for a single version 1/97 – 6/99

Figure 2: Bug Counts Over Time for Open and Closed Source Web Servers

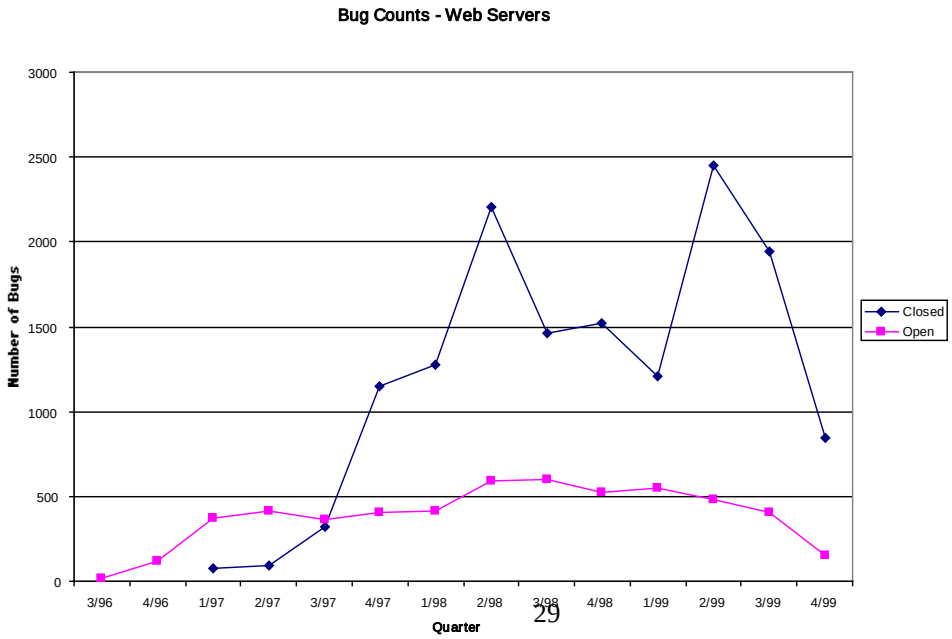

Figure 3: Bug Counts Over Time for Open and Closed Source Operating Systems

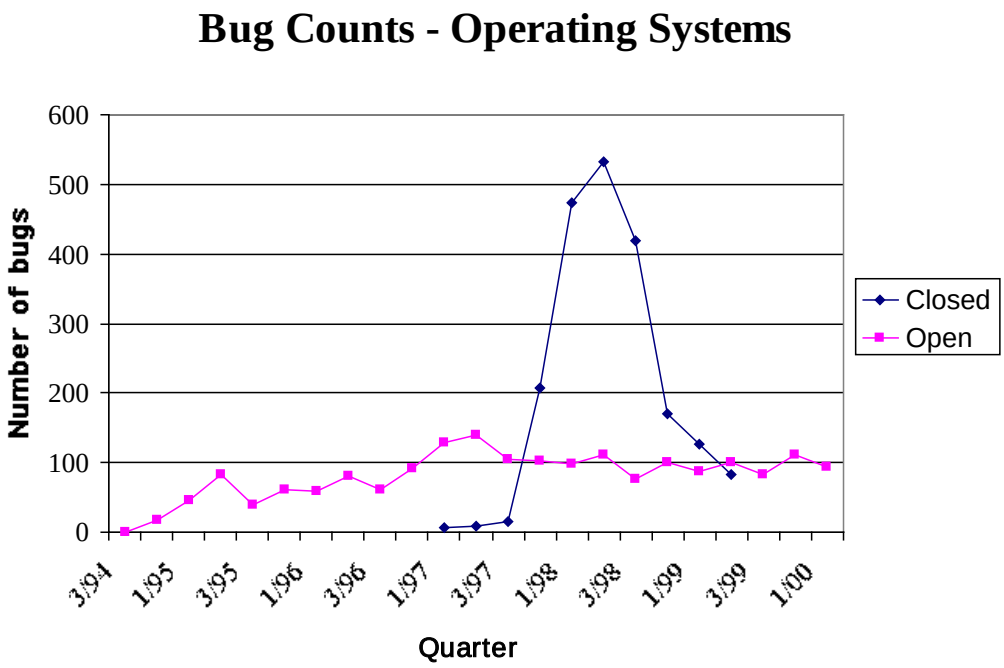

Figure 4: Baseline Hazard Rates for Open and Closed Source Web Servers

Figure 5: Baseline Hazard Rates for Operating Systems

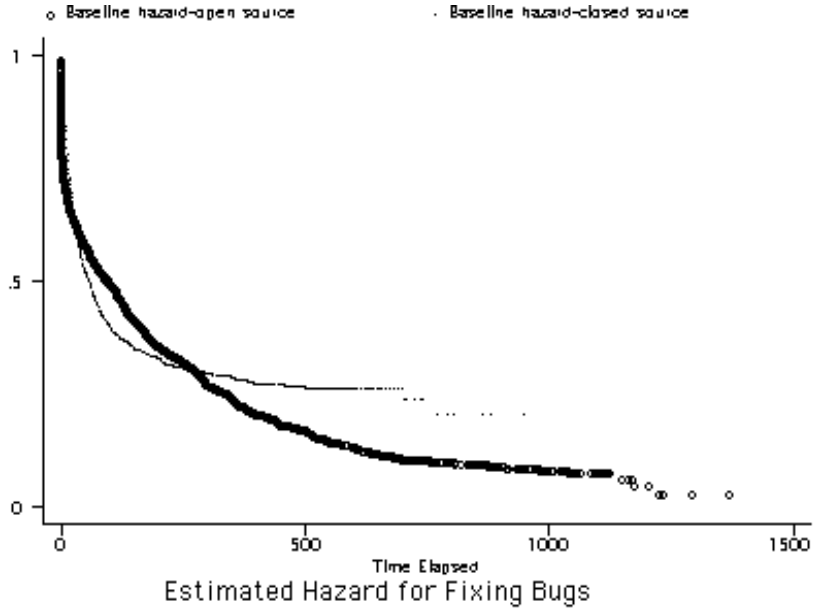

Figure 6: Baseline Hazard Rates for User Interfaces

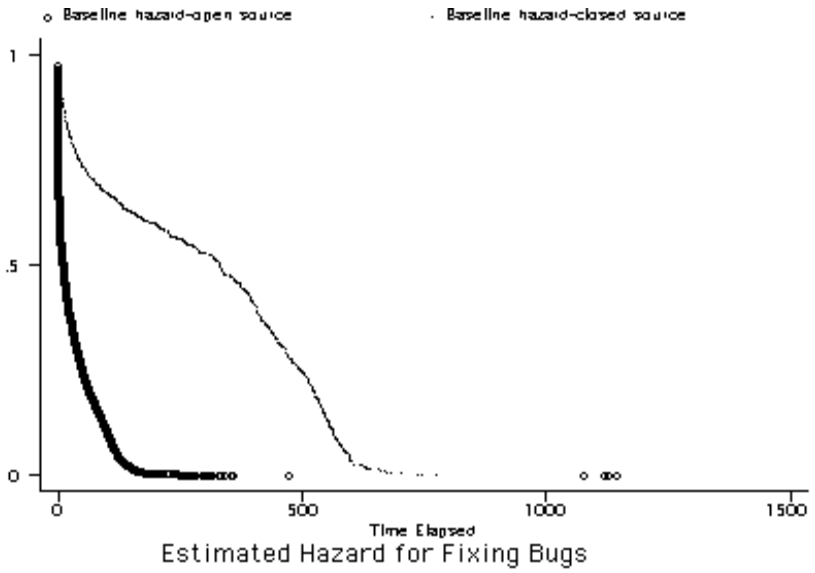

Table 2: Coefficients for Priority and Severity for All Programs

	Web Servers		Operating Systems		User Interfaces	
	Open	**Closed**	**Open**	**Closed**	**Open**	**Closed**
n	4853	14538	861	1834	1812	236
Severity	0.93*** (0.02)	0.86*** (0.01)	0.91* (0.05)	1.2*** (0.03)	0.88** (0.05)	1.1*** (0.03)
Priority	n/a	0.92*** (0.03)	0.88*** (0.05)	0.75*** (0.02)	n/a	0.95*** (0.03)

Table 3: Summary of Results

	Web Servers		Operating Systems		User Interfaces	
	Open	**Closed**	**Open**	**Closed**	**Open**	**Closed**
Bug resolution rate	Faster	Slower	Faster	Slower	Slower	Faster

Table 4: Some Successful Open Source Programs

Scripting Languages	Software Development Tools
• *Perl*	• *GNATS*
• *PHP*	• *LessTif*
• *Python*	• *GCC*
	• *Emacs*
Web server and Non-Front End Mail-Related Programs	• *RPM*
• *Apache*	• *GDB*
• *Sendmail*	• *CVS*
• *BIND*	Operating Systems
• *Fetchmail*	• *Linux*
• *SMTP*	• *FreeBSD*
• *POP*	• *OpenBSD*
• *IMAP*	• *NetBSD*